+

Advanced course

With

A. Derick

objectives

By the end of this presentation, participants will:

1. Understand the fundamentals of version control and the role of Git in modern development.
2. Gain proficiency in basic and advanced Git commands and workflows.
3. Learn how to effectively manage branches, resolve conflicts, and apply specific changes using cherry-picking.
4. Explore best practices for Git in DevOps, including CI/CD integration and GitOps.
5. Be equipped with practical knowledge to implement Git in collaborative and production environments.

Course outline

1. Introduction to Version Control

- **1.1.** What is Version Control?
- **1.2.** Types of Version Control Systems (VCS)
- **1.3.** Why Use Version Control?

2. Introduction to Git

- **2.1.** What is Git?
- **2.2.** Importance of Git in Modern Development

3. Git Basics

- 3.1. Installing and Configuring Git
- 3.2. Core Concepts
- 3.3. Basic Git Commands

4. Working with Branches

- 4.1. Understanding Branches
- 4.2. Creating and Managing Branches
- 4.3. Merging Branches
- 4.4. Rebasing
- 4.5 cherry-picking

5. Working with Remote Repositories

- 5.1. Introduction to Remote Repositories
- 5.2. Synchronizing with Remotes
- 5.3. Cloning and Forking

6. Advanced Git Features

- 6.1. Stashing Changes
- 6.2. Rewriting History with Interactive Rebase
- 6.3. Working with Submodules

7. Best Practices

- 7.1. Writing Good Commit Messages
- 7.2. Version Control Best Practices
- 7.3. Backup and Recovery

8. Conclusion

- Objective: Recap the key takeaways from the presentation and address any remaining questions.

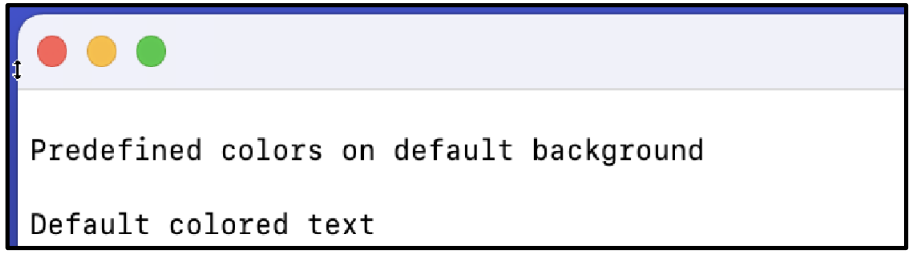
- 8.1. Summary of Key Takeaways
- 8.2. Q&A Session and hands-on project

9. References and Resources

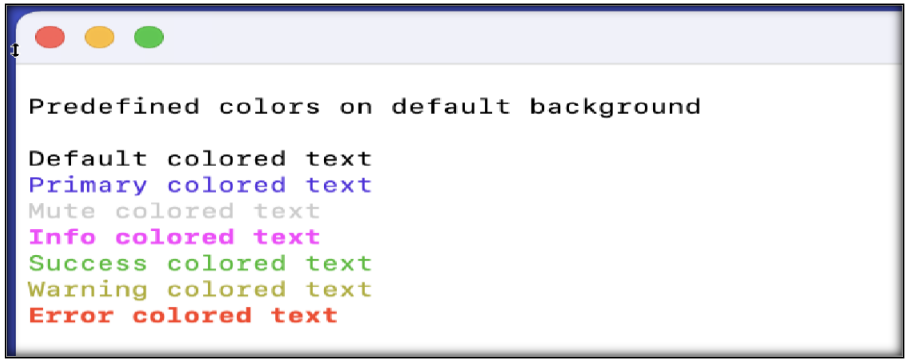
- 9.1. Suggested Reading and Tutorials
- 9.2. Tools and Plugins

1.1. What is Version Control?

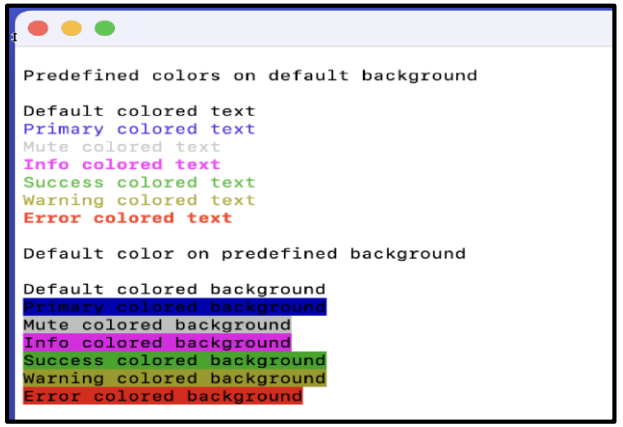
Version control is a system that tracks changes to files or sets of files (folder or project) over time. It allows multiple people to collaborate on a project, track changes, and revert to previous versions if needed.



Original file



First change



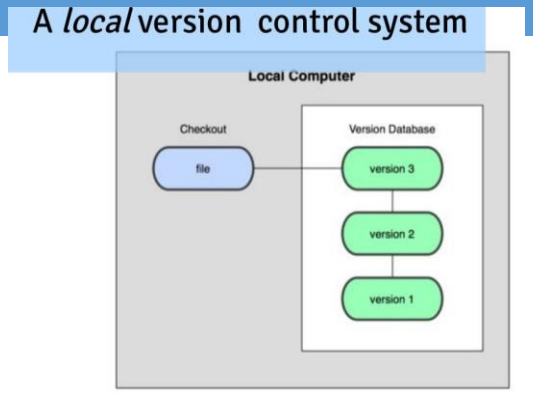
Second change

Purpose: Helps manage the evolution of a project, ensuring that changes are documented, reversible, and collaborative.

1.2. Types of Version Control Systems (VCS)

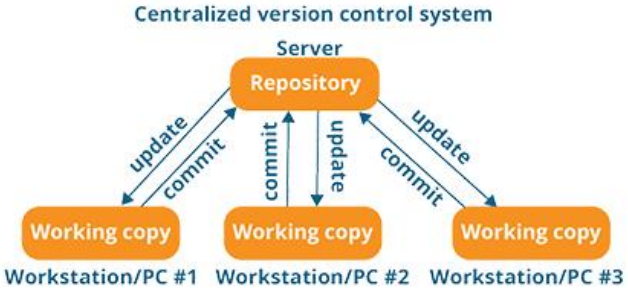
1. Local Version Control:

- Simple, stores changes on a local disk.
  - Limited to a single user's machine.



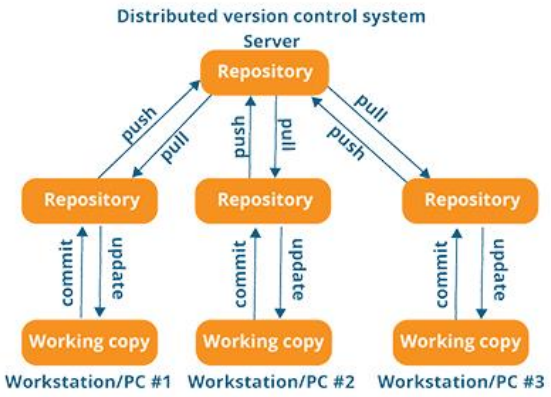
2. Centralized Version Control (CVCS):

- All versions are stored on a central server (e.g., Subversion).
  - Pros: Central management, easier to control access.
  - Cons: Single point of failure.



3. Distributed Version Control (DVCS):

- Each user has a full copy of the repository (e.g., Git, Mercurial)
  - Pros: No single point of failure, offline work possible.
  - Cons: More complex to manage, especially for beginners.



1.3. Why Use Version Control?

- **Collaboration:** Allows multiple developers to work on the same codebase without overwriting each other's work.

- **History Tracking:** Keeps a record of every change, who made it, and why.

- **Backup and Recovery:** You can easily restore previous versions if something goes wrong.

- **Branching and Merging:** Developers can work on different features or fixes in isolation and then merge them together.

2. Introduction to Git

2.1. What is Git?

Git is a Distributed Version Control System (DVCS) that allows developers to track changes, manage versions, and collaborate on code. Created by Linus Torvalds in 2005 for Linux kernel development

2.2 why choose Git

- **Widely Adopted:** Git is the most popular version control system in use today.

- **Open Source Contribution:** Essential for contributing to open-source projects, with platforms like GitHub built around Git.

- **Open Source Contribution:** Essential for contributing to open-source projects, with platforms like GitHub built around Git.

3. Git Basics

3.1 Installing Git

- **Windows:** Download and install from **(https://git-scm.com/download/win).**
- **macOS:** Install via Homebrew **(`brew install git`)** or download from the official site.
- **Linux:** Install via the package manager (e.g., `**sudo apt-get install git**` on **Ubuntu**).

3.2 configuring Git

- **Username and Email:** Set global user identity:

git config --global user.name "Your Name“ ------- > to set global user name

git config --global user.email "you@example.com" ------- > to set global user email

- **Editor:** Set your preferred text editor for Git:

git config --global core.editor "code --wait" ------- > visual studio is set as default here

- **Viewing Configuration:**

git config --list

3.2. Core Concepts

1. **working directory:** this in the initialized git repository in you local machine
2. **staging area:** stores information about what will go into your next commit
3. **local directory**: it information about all committed files
4. **remote repository:** A repository hosted on a server, often used for collaboration.
5. **Commits:** A snapshot of your project at a specific point in time, consisting of staged changes
6. **History:** Git logs all commits, allowing you to view the project's history.
7. **Branches:** Independent lines of development

3.3. Basic Git Commands

- Initialize a new repository:

```
git init
```

- Tracking changes

```
git status
```

- Adding unstaged files to the staging area

```
git add <fil_name>      to add a particular file
git add .               To add all files
```

- - Commit changes to the local directory:

```
git commit –m  <commit message>
git commit –am  <commit message>   to commit all file without staging
```

- View commit history:

```
git log --oneline
```

4. Working with Branches

4.1. Understanding Branches

- Definition: Branches are separate lines of development, allowing multiple versions of a project to coexist.

- Purpose: Facilitates parallel development, letting developers work on different features or fixes independently.

git branch <branch-name> **to create a branch**

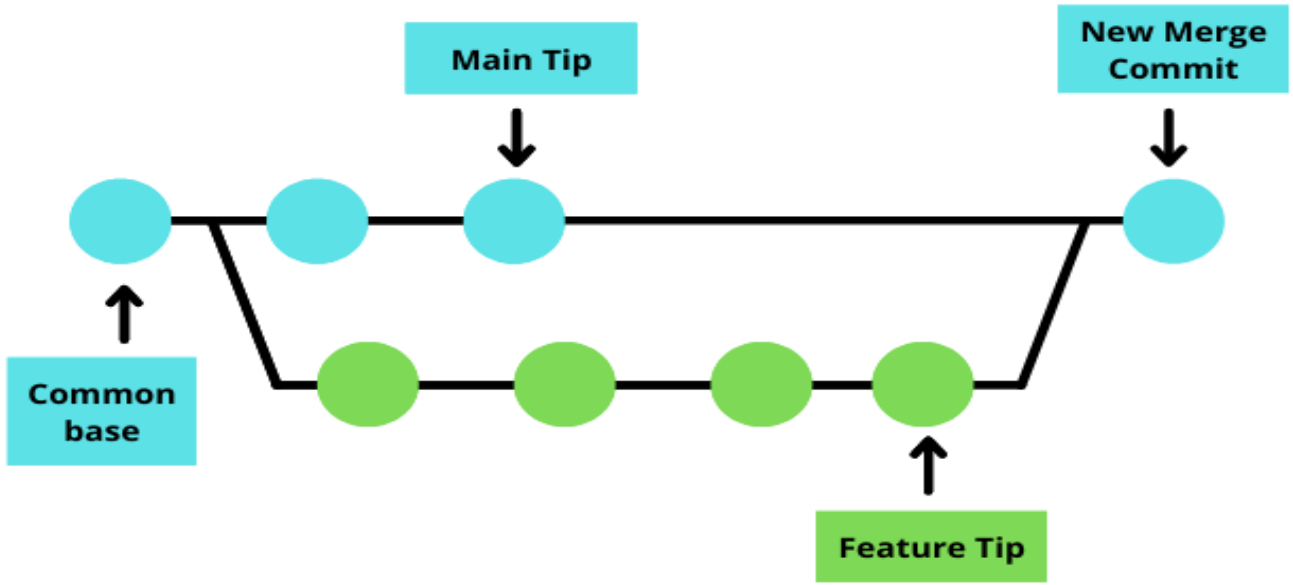
git checkout <branch-name> **to switch to a branch**

git checkout -b <branch-name> **to create and switch to a branch**

git branch **to view all the branches**

4.3. Merging Branches

Merging is the process of integrating changes from one branch into another.



How to merge future branch into the main branch

1. Switch to the main branch — **git checkout main**

2. Merge future branch into main — **git merge feauture**

`short form` — `git merge main feature`

4.4. Rebasing

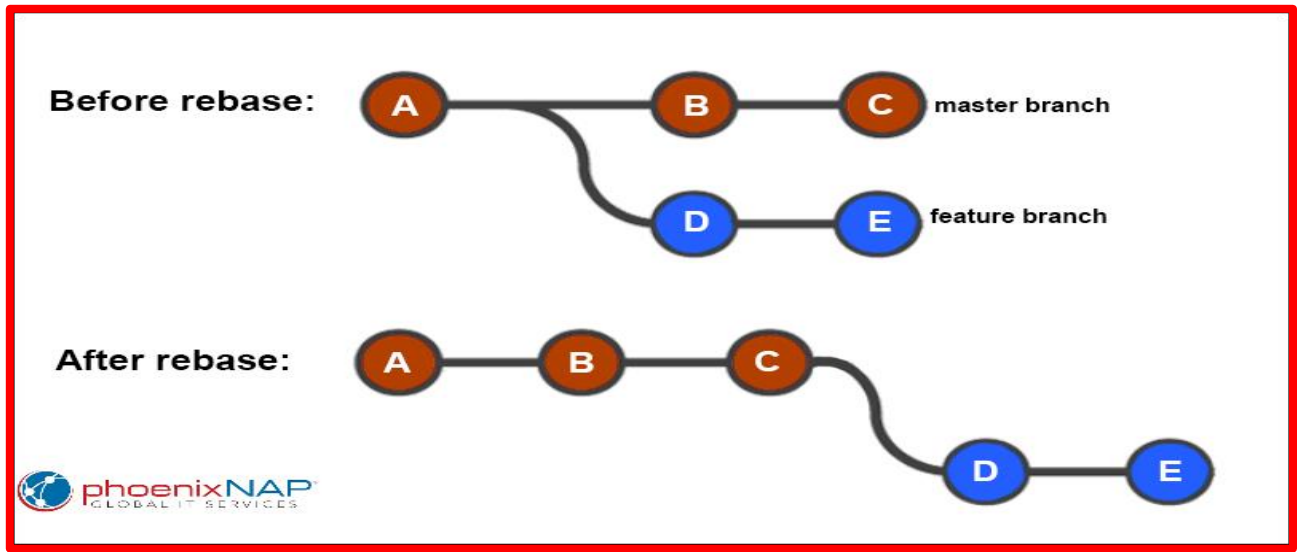
- What is Rebasing?

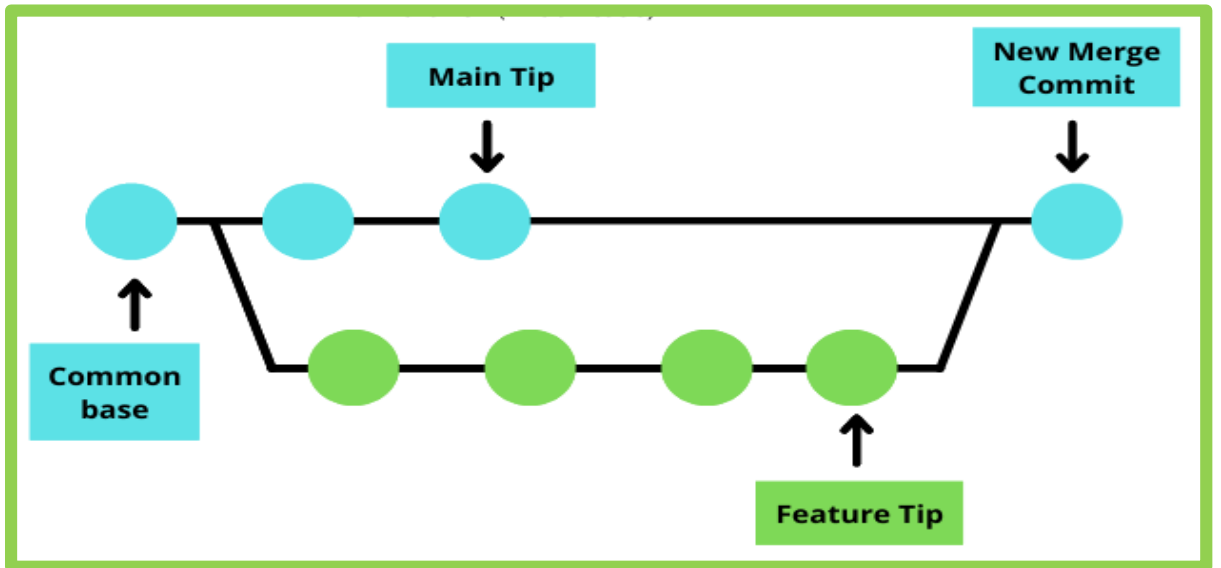
- Rebasing rewrites the commit history, creating a linear sequence of commits.

Rebasing vs. Merging:

- Merge: Combines two branches while preserving their history and creating a new commit.

- Rebase: Moves all commits of a branch to another branch, making the history appear linear. No new commit is created





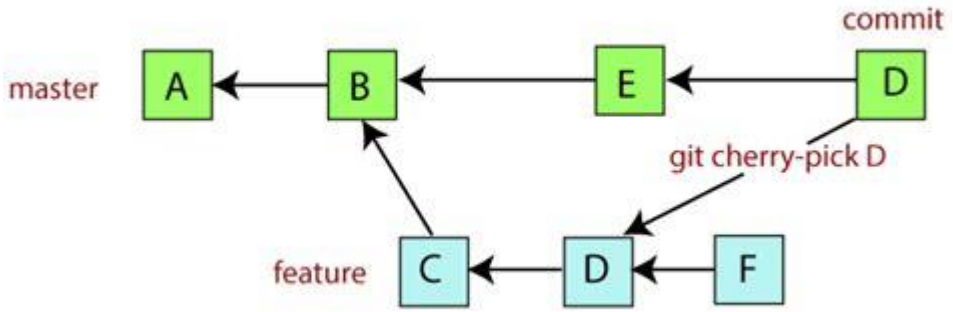
How to Rebase feature branch into main branch

1. Switch into the feature branch **git checkout <branch-name>**

2. Rebase into main **git rebase main**

Git cherry-picking

Cherry-picking allows you to apply a specific commit from one branch to another without merging the entire branch.



steps to cherry-pick

git checkout main

git cherry-pick <commit_hash>

5. Working with Remote Repositories

- **Remote repositories** are hosted versions of your project, typically on platforms like GitHub, GitLab, or Bitbucket.
- **Purpose:** Facilitates collaboration, backup, and deployment.

- Remote repositories are hosted versions of your project, typically on platforms like GitHub, GitLab, or Bitbucket.

GitLab

ATLASSIAN

Bitbucket

5.2. Synchronizing with Remotes

- **Pull changes from the remote and merge**

  git pull [remote] [branch]

  [remote] = remote directory

  Example: git pull origin main

  [branch] = the branch you want to pull from

- **Fetch changes from the remote without merging**

  git fetch **to merge run** **git merge**

- **Push local commits to the remote repository:**

  git push origin <branch>

- Create a local copy of a remote repository:

  git clone <repository_url>

6. Advanced Git Features

6.1. Stashing Change:

Stashing temporarily saves your uncommitted changes, allowing you to switch branches or work on something else without losing your current work.

To stash a change

1. **git add <file_name>** — **add the changes made**
2. **git stash** — **stash the changes**
3. **git stash list** — **list all stashes**
4. **git stash apply** — **apply last stash without delete**
5. **git stash clear** — **deletes all the stash**

git blame <file_name>

shows who made changes to each line of a file and when. It’s useful for tracking the origin of specific code lines.

Squashing [git rebase -i head~n]

squashing helps you to combine many comitts into 1 commit. n in the command is the number of commits you want to squash

git diff <commit1> <commit2>

helps you to get the difference between two commits

git restore --staged <file>

to unstage a file

6.2 Rewriting commit messages

git rebase -i <commit> — **change commit message for a particular commit**

git commit - -amend — **change commit message of the last commit**

6.3. Working with Submodules

Submodules allow you to include and track other Git repositories within your project

1. **git submodule add <repository-url>**
2. **git submodule init**
3. **git submodule update**

6.4 Git Tagging

Tags are references to specific points in Git history, often used for marking releases. They are ways to name specific commits for releases

- **Lightweight Tag:**
  git tag <tag-name>
- **annotated Tags:** Includes a message, author, and date, and is stored as a full object in the Git database.
  git tag –a <tag-name> -m “tag message”

git push origin <tag_name> **to push a tag to the remote directory**
git push - -tags **to push all tags**

git tag <tag_name> **to view info about a tag**
git tag **to get all tags**

7. Git Best Practices

7.1. Commit Messages

- Best Practices:

  - Clear and Concise: Summarize the changes in a clear, concise manner.

  - Imperative Mood: Use the imperative mood (e.g., "Fix bug" instead of "Fixed bug").

  - Structure:

    - Title: One-line summary of changes.

    - Body (optional): Detailed explanation of the changes, rationale, and any related issues.

  - Example: git commit -m "Fix login issue on mobile view"

7.2. Branch Naming Conventions

- Best Practices:
  - Descriptive: Use descriptive names that convey the purpose of the branch.
  - Consistency: Follow a consistent naming pattern.
  - Common Patterns:
    - Feature Branches: `feature/<feature-name>`
    - Bugfix Branches: `bugfix/<issue-number>`
    - Hotfix Branches: `hotfix/<issue-number>`
    - Release Branches: `release/<version-number>`
  - Example:
    git checkout -b feature/user-authentication

8.1. Recap of Key Points

- Version Control: Importance and benefits of using version control systems.

- Git Basics: Core concepts and basic commands.

- Branches and Merging: Creating, managing, and merging branches.

- Advanced Features: Stashing, rebasing, cherry-picking, and more.

- Workflows: Common Git workflows and their use cases.

- Best Practices: Commit messages, branch naming

Q & A

PROJECT